# 18장. 리플렉션과 어노테이션

# 18.1. 리플렉션

18.1.1. 리플렉션 이해

리플렉션이란?

- 리플렉션은 런타임시 프로그램의 구조(객체, 함수, 프로퍼티)를 분석해 내는 기법
- kotlin-reflect.jar 라는 라이브러리에서 제공

클래스 타입과 레퍼런스

- 런타임시 동적인 클래스 분석
- 클래스에 대한 정보를 클래스 Reference , 클래스 Reference가 대입되는 곳은 클래스 타입
- 클래스 타입은 KClass<*> 로 표현하며 이곳에 대입되는 클래스 Reference는 “클래스명::class” 로 표현

```
val myVal: KClass<*> = String::class

fun myFun(arg: KClass<*>){

}
```

# 18.1. 리플렉션

- KClass<*> 은 모든 타입의 클래스 Reference 도 대입이 가능
- 특정 타입의 클래스 Reference만 대입되어야 한다면 Kclass<클래스명>
- 자바 클래스의 Reference는 .java 를 추가

```
val myVal: KClass<String> = String::class

//val myVal2: KClass<String> = Double::class//error

val myVal3: Class<*> = String::class.java
```

# 18.1. 리플렉션

18.1.2. 클래스 레퍼런스

클래스 정보 분석

- val isAbstract: Boolean : 클래스 Reference 가 abstract 로 선언되었는지 판단
- val isCompanion: Boolean : 클래스 Reference 가 companion 로 선언되었는지 판단
- val isData: Boolean : 클래스 Reference 가 data 로 선언되었는지 판단
- val isFinal: Boolean : 클래스 Reference 가 final 로 선언되었는지 판단
- val isInner: Boolean : 클래스 Reference 가 inner 로 선언되었는지 판단
- val isOpen: Boolean : 클래스 Reference 가 open 로 선언되었는지 판단
- val isSealed: Boolean : 클래스 Reference 가 sealed 로 선언되었는지 판단

생성자 분석

- val constructors: Collection<KFunction<T>> : 모든 생성자 정보
- val <T : Any> KClass<T>.primaryConstructor: KFunction<T>? : 주생성자 정보

# 18.1. 리플렉션

클래스 프로퍼티 분석

- val <T : Any> KClass<T>.declaredMemberProperties: Collection<KProperty1<T, *>> : 확장 프로퍼티를 제외한 클래스에 선언된 모든 프로퍼티 리턴
- val <T : Any> KClass<T>.memberProperties: Collection<KProperty1<T, *>> : 확장 프로퍼티를 제외한 클래스와 상위 클래스에 선언된 모든 프로퍼티 리턴
- val <T : Any> KClass<T>.declaredMemberExtensionProperties: Collection<KProperty2<T, *, *>> : 클래스에 선언된 확장 프로퍼티 리턴
- val <T : Any> KClass<T>.memberExtensionProperties: Collection<KProperty2<T, *, *>> : 상위 클래스 및 현 클래스의 확장 프로퍼티 리턴

클래스 함수 분석

- val KClass<*>.declaredMemberFunctions: Collection<KFunction<*>> : 확장 함수를 제외한 클래스에 선언된 모든 함수 리턴
- val KClass<*>.memberFunctions: Collection<KFunction<*>> : 확장 함수를 제외한 클래스와 상위 클래스에 선언된 모든 함수 리턴
- val KClass<*>.declaredMemberExtensionFunctions: Collection<KFunction<*>> : 클래스에 선언된 확장 함수 리턴
- val KClass<*>.memberExtensionFunctions: Collection<KFunction<*>> : 상위 클래스 및 현 클래스의 확장 함수 리턴

# 18.1. 리플렉션

18.1.3. 함수 레퍼런스와 프로퍼티 레퍼런스

함수 레퍼런스 분석

- 함수 Reference는 “::함수명” 을 이용, 함수는 KFunction<*> 타입으로 이용

```
fun myFun(){ }

class MyClass {
    fun myFun2() { }
}

val funReference: KFunction<*> = ::myFun

val funReference2: KFunction<*> = MyClass::myFun2
```

- val name: String : 함수이름
- val parameters: List<KParameter> : 함수에 선언된 모든 매개변수
- val returnType: KType : 함수의 리턴 타입

# 18.1. 리플렉션

고차함수 호출시 이용

```
fun isOdd(x: Int): Boolean = x % 2 != 0

fun reflectionFun(argFun: (Int) -> Boolean){
    println("result: ${argFun(3)}")
}

fun main(args: Array<String>) {
    reflectionFun { n -> isOdd(n) }
    reflectionFun(::isOdd)
}
```

```
fun myFun(no: Int): Boolean {
    return no > 10
}
fun main(args: Array<String>) {
    val array = arrayOf<Int>(10, 5, 30, 15)

    println("람다함수 이용...")
    array.filter { it > 10 }
            .forEach { println(it) }

    println("함수 Reference 이용....")
    array.filter(::myFun)
            .forEach { println(it) }

}
```

# 18.1. 리플렉션

프로퍼티 레퍼런스 분석

- 프로퍼티 Reference는 "::프로퍼티명", 타입은 KProperty<*> 와 KMutableProperty<*>

```
val myVal: Int = 10

var myVar: Int = 10

val referenceVal1: KProperty<*> = ::myVal

val referenceVal2: KProperty<*> = ::myVar

//val referenceVal3: KMutableProperty<*> = ::myVal//error

val referenceVal4: KMutableProperty<*> = ::myVar

fun main(args: Array<String>) {
    println(::myVal.get())

    ::myVar.set(30)
    println(::myVar.get())
}
```

# 18.1. 리플렉션

Kproperty

- val getter: Getter<R> : get() 함수의 정보
- val isConst: Boolean : const으로 선언된건지 판단
- val isLateinit: Boolean : lastinit으로 선언된건지 판단
- val isAbstract: Boolean : abstract로 선언된건지 판단
- val isFinal: Boolean : finale로 선언된건지 판단
- val isOpen: Boolean : open 으로 선언된건지 판단
- val name: String : 프로퍼티 이름 획득
- val returnType: KType : 프로퍼티 타입 획득
- fun call(vararg args: Any?): R : 프로퍼티 get() 호출

KMutableProperty

- val setter: Setter<R> : set() 에 대한 정보

# 18.2. 어노테이션

18.2.1. 어노테이션 작성 및 이용

- 어노테이션은 클래스, 함수, 프로퍼티 선언 앞에 추가되는 구문으로 @로 시작하는 구문

- 컴파일러에게 코드 문법 에러를 체크하기 위한 정보를 제공
- 개발 툴이나 빌더에게 코드 자동 추가를 위한 정보
- 실행시 특정 기능을 실행하기 위한 정보

- 어노테이션은 annotation 예약어로 만들어지는 클래스
- 객체생성 불가
- 실행영역을 가질수 없다.

```
annotation class TestAnnotation

annotation class TestAnnotation2 { }//error

fun main(args: Array<String>) {
    val obj: TestAnnotation = TestAnnotation()//error
}
```

# 18.2. 어노테이션

```
annotation class TestAnnotation

@TestAnnotation
class Test {

    @TestAnnotation
    val myVal: String = "hello"

    @TestAnnotation
    fun myFun() {

    }
}
```

```
annotation class TestAnnotation

class Test @TestAnnotation constructor(){
    @TestAnnotation
    val myVal: Int=10

    var myVal2: Int = 10
        @TestAnnotation set(value) { field = value }

    val myFun = @TestAnnotation{

    }
}
```

# 18.2. 어노테이션

18.2.2. 어노테이션 설정

데이터 설정

- 주생성자를 이용해 데이터가 설정
- val이 추가되어야 하며 var은 허용되지 않는다.

```
annotation class TestAnnotation(val count: Int)
class Test {
    @TestAnnotation(count=3)
    fun some(){
        println("some.....")
    }
}
fun main(args: Array<String>) {
    val obj: Test = Test()
    val methods = Test::class.java!!.methods

    for(method in methods){
        if(method.isAnnotationPresent(TestAnnotation::class.java)){
            val annotation=method.getAnnotation(TestAnnotation::class.java)
            val count = annotation.count
            for(i in 1..count){
                obj.some()
            }
        }
    }
}
```

# 18.2. 어노테이션

허용 데이터 타입

- 자바의 기초 타임(Int, Long etc.)
- string
- classes(Foo::class)
- enums
- other annotations
- array of the types listed above

```
annotation class TestAnnotation1(val count: Int)

annotation class TestAnnotation2(val otherAnn: TestAnnotation1, val arg1: KClass<*>)

class User

//annotation class TestAnnotation3(val user: User)//error

@TestAnnotation1(10)
@TestAnnotation2(TestAnnotation1(20), String::class)
class Test { }

const val myData: Int = 10
@TestAnnotation1(myData)
class Test2 { }
```

# 18.2. 어노테이션

어노테이션 선언 옵션

- @Target : 어느 곳에 사용하기 위한 annotation인지를 명시하기 위해서 사용 (classes, functions, properties, expressions)
- @Retention : annotation을 컴파일한 클래스에 보관할지, 런타임시 Reflection에 의해 접근할수있는지에 대한 설정. (source, binary, runtime)
- @Repeatable : 이 annotation을 한곳에 반복 사용이 가능하게 설정.
- @MustBeDocumented : annotatoin을 api에 포함시켜야 하는지, api document 문서에 포함되어야 하는지에 대한 설정

# 18.2. 어노테이션

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
        AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.SOURCE)
annotation class TestAnnotation

@TestAnnotation
class Test {
    @TestAnnotation constructor(no: Int){}//error

    @TestAnnotation//error
    val myVal: Int = 10

    @TestAnnotation
    fun myFun(@TestAnnotation no: Int) {
        val result = @TestAnnotation if (no>10){
            10
        }else {
            0
        }
    }
}
```

# 18.2. 어노테이션

### 18.2.3. 어노테이션 적용 대상 지정

- file
- property (annotations with this target are not visible to Java)
- field
- get (property getter)
- set (property setter)
- receiver (receiver parameter of an extension function or property)
- param (constructor parameter)
- setparam (property setter parameter)

```
class Test constructor(@param: TestAnnotation var email: String){

    @get:[TestAnnotation TestAnnotation2]
    var no: Int=10

    @property: TestAnnotation
    var name: String = "kkang"

    @field: TestAnnotation
    var age: Int = 30

    @setparam: TestAnnotation
    var phone: String= "0100000"
}
fun @receiver: TestAnnotation Test.myFun(){ }
```

# 18.2. 어노테이션

18.2.4. 자바 어노테이션 이용

- 자바로 선언된 어노테이션을 코틀린에서 사용이 가능
- 데이터 설정이 되어야 한다면 데이터의 순서적인 문제
- 데이터를 대입시킬 때 꼭 이름을 명시

```
public @interface JavaAnnotation {
    int intValue();
    String stringValue();
}
```

```
annotation class KotlinAnnotation(val no: Int, val name: String)

@KotlinAnnotation(10, "kkang")
//@JavaAnnotation(10, "kkang")//error
@JavaAnnotation(intValue = 10, stringValue = "kkang")
class Test { }
```

# 18.2. 어노테이션

- 자바 어노테이션의 함수명이 value 로 되어 있다면 이때는 이름을 명시하지 않아도 된다.

```
public @interface JavaAnnotation2 {
    int value();
    String strValue();
}
```

```
@JavaAnnotation2(10, strValue = "kkang")
class Test { }
```

- 데이터가 배열로 대입되어야 하는 경우 배열의 함수가 value 로 선언되었다면 데이터만 나열, value 함수가 아닌 경우는 arrayOf() 를 이용

```
public @interface JavaAnnotation3 {
    int[] value();
    String[] strVale();
}
```

```
@JavaAnnotation3(10, 20, strVale = arrayOf("kkang", "kim"))
class Test { }
```